

EXHIBIT K

Efficiently Tracking Application Interactions using Lightweight Virtualization¹

Yih Huang
George Mason University
4400 University Drive
Fairfax, Virginia, 22030
+1-(703)-993-9681
huangih@gmu.edu

Angelos Stavrou
George Mason University
4400 University Drive
Fairfax, Virginia, 22030
+1-(703)-993-3772
astavrou@gmu.edu

Anup K. Ghosh & Sushil Jajodia
George Mason University
4400 University Drive
Fairfax, Virginia, 22030
+1-(703)-993-{3531, 4776}
{aghosh1, jajodia}@gmU.edu

ABSTRACT

In this paper, we propose a general-purpose framework that harnesses the power of lightweight virtualization to track applications interactions in a scalable and efficient manner. Our goal is to use our framework for application auditing, intrusion detection, analysis, and system recovery from both malicious attacks and programmatic faults. In our framework, we construct each virtualized environment (VE) in a novel way that limits the scope and type of application events that need to be monitored.

Our approach maintains the VE and system integrity, having as primarily focused on the interactions among VEs and system resources including the file system, memory, and network. Only events that are pertinent to the integrity of an application and its interactions with the operating system are recorded. We attempt to minimize the system overhead both in terms of system events we have to store and the resources required. Even though we cannot provide application replay, we keep enough information for a wide range of other uses, including system recovery and information tracking among others. As a proof of concept, we have implemented a prototype based on OpenVZ[35], a lightweight virtualization tool. Our preliminary results show that, compared to state-of-the-art event recording systems, we can reduce the amount of event recorded per application by almost an order of magnitude.

Categories and Subject Descriptors

C.4.0 [Computer System Implementation]: General;
D.4.6 [Operating Systems]: Security and Protection.

General Terms

Measurement, performance, experimentation, security.

Keywords

Lightweight, virtualization, interactions, tracking, applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VMSEC'08, October 31, 2008, Fairfax, Virginia, USA.

Copyright 2008 ACM 978-1-60558-298-6/08/10...\$5.00.

1. INTRODUCTION

The prevailing uses of computers in our everyday life give rise to many new challenges, including system security, data protection, and intrusion analysis. Towards that direction, recent research has provided two new powerful tools: 1) virtualized execution environments and 2) the monitoring, inspection, and/or logging of system activities via system call monitoring. In this paper, we introduce a novel general-purpose framework that unifies the two mechanisms and models as transactions the interactions of applications harnessing the resource efficiency and isolation provided by lightweight virtualization environments (VEs). Our framework attempts to lay a strong foundation that can be employed to stop cross-application contamination, perform corruption analysis, or file and system recovery after an attack among others. The proposed approach leverages the powers the two mechanisms but also mutually addresses the weakness of each other: on the virtualization side the need to protect the integrity of virtualized environments by monitoring, and by using virtualization we reduce the number of activities that we need to monitor making it manageable. The result is a framework that protects the integrity of virtualized environments with very low overhead and also tracks whole system by focusing on interactions among virtualized environments. We treat the operating system as an intermediate layer that we protect but do not closely monitor giving emphasis on the applications and their interactions. Before describing our framework, we provide a small summary of the features of the two constituent technologies.

Virtualization can be used to encapsulate applications in a dedicated computing environment, complete with its own network identity, process space, and file system. With the isolation of execution environments enforced by the virtualization layer, it is sometimes assumed that corruptions of or attacks against one virtualized application execution will not spill over to other virtual environments. Applications, however, have legitimate reasons, in many cases the necessity, to interact with one another, regardless whether they are executed in virtualized environments. A web browser, for instance, needs to interact with a remote web server. In this case, the medium of the interaction is typically network sockets. On a personal computer, the user can run an email client and office suite in two virtual machines in order to isolate the two applications. Still, a user commonly needs to save an office document attachment from email and open the document in the office application software suite. This constitutes a data

¹This work was partially supported by Air Force Office of Scientific Research under grant FA9550-07-1-0527 and by National Science Foundation under grants CT-0716567, CT-0627493, and IIS-0430402.

exchange. In general, applications executed in isolated virtual environments must still interact and communicate with one another, creating the prospects that corruptions/intrusions in one environment *can* propagate to others.

Another powerful approach to enhancing system security, reliability, and recoverability involves monitoring, logging, and analyzing system activities. Monitoring system activities in real time has the potential to intercept and stop ill-intended actions (such as intrusions) from happening in the first place. Analyzing logs provides the opportunities to detect corruptions or intrusions afterward, identify corrupted system and data elements, isolate and remove corruptions, recover lost data, restore the system to a clean state, and reveal attack paths – inbound and outbound. An important and challenging tradeoff of such an approach is the granularity of the events to monitor. On one extreme, the execution of every machine instruction can be inspected and logged to obtain the most detailed knowledge of information flow. The cost of doing so could be prohibitively high for many applications. On the other extreme, many administrators use only system and application logs in investigation and recovery. These logs usually do not provide sufficient information for analysis and are susceptible to tampering by attackers. Some intrusion detection and system recovery solutions choose to monitor system calls for anomalous events. Even at the system call level, a modern computer can generate large amounts of events in short periods of time. Furthermore, a particular solution usually fixates on a chosen granularity of system activities regardless of context.

To that end, we propose a general-purpose framework that unifies virtualization and logging in a transaction framework for applications of interest to enable protection, monitoring, detection, analysis, and recovery. In our framework, each virtualized environment (VE) is constructed in a novel way that limits the scope of events to be monitored to protect its integrity, leaving events monitoring to primarily focus on the interactions among VEs and system resources including the file system, memory, and network. The integrity of a VE is still monitored and protected but with drastically reduced overhead. Specifically, our framework, termed Journaling Computing System (JCS), makes the following novel contributions.

- We model as *transactions* all interactions among virtualized applications executions, and interactions with remote hosts/servers. The granularity of interaction events that are monitored are determined by specific requirements (for system recovery, intrusion analysis, and so on). Each individual event is treated as a transaction including only one operation. Such a transaction is called *atomic*.
- Isolation and Monitoring of VEs. A large body of research has been dedicated to VE monitoring and inspections in order to protect its integrity. JCS, in contrast, focuses on those activities that cause interactions with other VEs and remote hosts. By using a branching copy-on-write file system that provides views of a common read-only branch, the overhead of monitoring VE integrity *is restricted only to persistent files*, thus reducing the monitoring cost. We will present how this is achieved by file namespace unifications. Without the burden of protecting VE integrity with comprehensive monitoring, we focus on the interactions among VEs and remote hosts for whole-system protection.

- Towards summarized transactions. Just like a database transaction may include multiple operations, in JCS a set of atomic transactions can be combined to produce summarized transactions. A good example is to summarize reads and writes on a file according to the open-close cycles. Alternatively, one can summarize as on ST the effects of a process on a file during its lifespan. Currently, we are in the process of formalizing a lossless summarization method to help reduce the size of event logs.

2. RELATED WORK

IBM first introduced computer virtualization as a product in 1972 [26]. It has in recent years experienced a powerful revival. A non-exhaustive list of present virtualization technologies includes VMware products [27], Xen [28], User Mode Linux [29], KVM [30] and VirtualBox [31]. Virtualization lends itself to “out-of-box” views of a computing system. By virtualizing the computing system under protection, the monitor can be executed outside the system and are hence isolated from the corruption within the system. Furthermore the monitoring software can be placed in separate virtual machines, avoiding the use of dedicated monitoring servers as in the case of Forensix. These critical properties have been extensively studied in recent research and used for intrusion detection [1, 2, 3, 4], intrusion forensics [5, 6], integrity monitoring [7, 8], trusted computing [9], system analysis [10, 11], and honey-pots and honey-clients [12, 13, 14, 15, 16]. Typically in such research, a VM under protection is *persistent*, that is, it is created and configured a prior to run an application or provide a service and is expected to be in use for long periods of time. Only [14] and [16] create virtual machines on demand and use them in a transient manner. The resultant virtual machines, however, do not have the isolation and monitoring properties of JCS VEs and must be inspected by traditional ways to determine corruption.

The research discussed above use either full virtualization or para-virtualization. We point out that while running complete operating systems in VMs provides isolation and opportunities of out-of-box inspection, it does not address the issue of the amounts of system activities that need attention --- an operating system produces large numbers of activities, whether executed in a virtual machine or directly on a physical host. With lightweight virtualization, virtualized execution environments share the same OS kernel of the underlying host, yet equipped with separate and isolated file systems, process namespaces, and network protocol stacks. It provides isolation without the overhead of fully virtualized guest operating systems. Examples of lightweight virtualization technologies include BSD Jail [31], Linux VServer [32], Zap [33], Solaris Zones [34], and OpenVZ [35]. Prior work in this direction primarily focuses on isolation of applications [32] and efficient service migration for high availability [33, 34, 35]. The JCS framework presented in this paper is a general approach that models as transactions the interactions among lightweight VEs, for the protection and recovery of the whole system.

There exists a lot of early research focused in monitoring application system calls to detect and react to abnormal behavior [17, 18]. This line of research has been recently revisited and extended to include the system call context, that is, the user-space call stacks leading to the system calls [19, 20]. In addition, system call logging has also been used for intrusion analysis and data recovery extensively [21, 22, 23, 24, 25]. These approaches typically adopt *comprehensive* logging: all system activities from all running process in a system are logged. A potential issue that

stems from comprehensive logging is the system and storage overhead that they involve especially for busy servers. To address this issue for instance, the Forensix system requires a dedicated backend machine for logging [22, 23]. Contrary to that, our approach attempts to avoid processing and storing large amount of logging data. In particular, by executing applications in isolated and monitored VEs, created on demand, we minimize the overhead requirements in order to monitor applications' integrity. To that end, we only monitor interactions among VEs, which are the focus of our whole system protection. The modeling of inter-VE interactions as high-level transactions is expected to further reduce the causality complexities --- the power of such reduction is the subject of our ongoing research.

3. SYSTEM OVERVIEW

JCS aims to provide a framework that unifies secure application execution in virtualized environments and the monitoring and logging of system events. We expect it serve as a strong foundation for protecting application integrity, system security, reliability, recoverability, data protection, and intrusion prevention/detection/analysis. It also aims to address the basic problems of these two general approaches: the protection of VE integrity and the amount of events needed to be monitored and logged. To achieve this goal, we model system events as transactions, provide ways to summarize transactions, and construct VEs in a unique way that drastically reduces the number of events that concern VE integrity. The result is a framework that focuses on inter-VE interaction events. Let us elaborate.

3.1 VE Interactions as Transactions

In JCS, interactions among VEs are modeled as transactions. It is a requirement of JCS that the underlying virtualization technologies prohibits processes running in different VEs from sharing memory, send signals or communicate with IPC facilities. Under this requirement, VEs interact with each other or remote hosts using the same mechanisms like machines in a distributed computing system: through data sharing or socket connections. We model such VE interactions as database transactions.

In a database model, a transaction is a partial order of reads and writes on a set of variables. The exact meaning of variables is defined by applications. The database enforces that operations in a transaction will either be committed or aborted. In the former case, all writes retains their results. In the latter, none of them does.

In JCS, all aspects of a computing system, including files, memory, and network communication events can be modeled as transactions on variables. For the file system and memory, every byte in a file or memory address space can be modeled as an individual variable. Alternately, we can use a higher level model where variables in transactions are byte intervals, corresponding to sectors on disk or memory pages. While the JCS model applies to all levels of granularity, practical considerations will guide the realization. Seeing exactly what bytes in a file are read or written by whom at what time requires monitoring the read/write system calls, a common practice in previous research. Seeing the

reads/writes on individual memory bytes requires intercepting every load/store machine instructions. While possible, this finest memory granularity will likely incur significant overhead unacceptable to some applications. At even higher level, an entire file or the whole virtual memory space of a process can be considered variables. In JCS, we go one step further: the combined virtual memory spaces of all processes in each VE are treated a single memory variable. One can see clearly that the emphasis is on VE-system and inter-VE interactions; we do not attempt to monitor the memory internals of VEs.

Modeling socket communications as transactions requires recognizing a special type of variable. Consider a scenario where we create a socket to connect to 64.236.16.20 on port 80 (one of the IP addresses of cnn.com), the socket becomes a means to write to and read from a variable named (64.236.16.20, 80), aka an endpoint in TCP/UDP terminology. Different from file-based or memory-based variables, a TCP-endpoint variable is *volatile*: different reads from the same variable may produce different results (consider "reads" on cnn.com at different times). In the JCS framework, communication activities are modeled as transactions involving volatile, TCP/UDP endpoint variables.

To implement commits and aborts requires a real-time monitor, which stops (or aborts) illegal transactions according to predefined policies and allows others to complete (be committed). It must be emphasized that we do not fix on any specific granularity of transaction semantics. If a file transaction is just a write system call, then aborting the transaction requires stopping the system call. If a file transaction is defined to include all the writes occurring in an open-close cycle, then aborting the transaction must restore the file to its state before the open operation. Aborting transactions on different type of variables involve different technologies. Aborting a socket-based transaction that includes multiple packet exchanges could involve a proxy server. Recent research on process space checkpoint and rollback can be considered as mechanisms for aborting memory transactions.

3.2 Application VEs Created on Demand

In JCS, applications are executed in Virtualized Environments (VEs). For a personal computer, web browsers, email clients, office suites, instant messengers, media players, and PDF readers among others are each run in their own VE. For a server, we will focus on applications that provide network services. We emphasize that in JCS every application instance is executed in its own dedicated VE. With a JCS-based PC, for example, every click on the Firefox icon creates a new VE comprising minimal system utilities/libraries and Firefox binaries. The Firefox application is then executed in the newly created VE and its window(s) displayed to the desktop. When the user terminates an application, the corresponding VE is terminated. More importantly, JCS VEs are constructed in a way that corruptions in the file system can be easily revealed without extensive monitoring. The primary focus of event monitoring is on the interactions among VEs and the system resources it uses, particular, interactions through data sharing and socket connections.

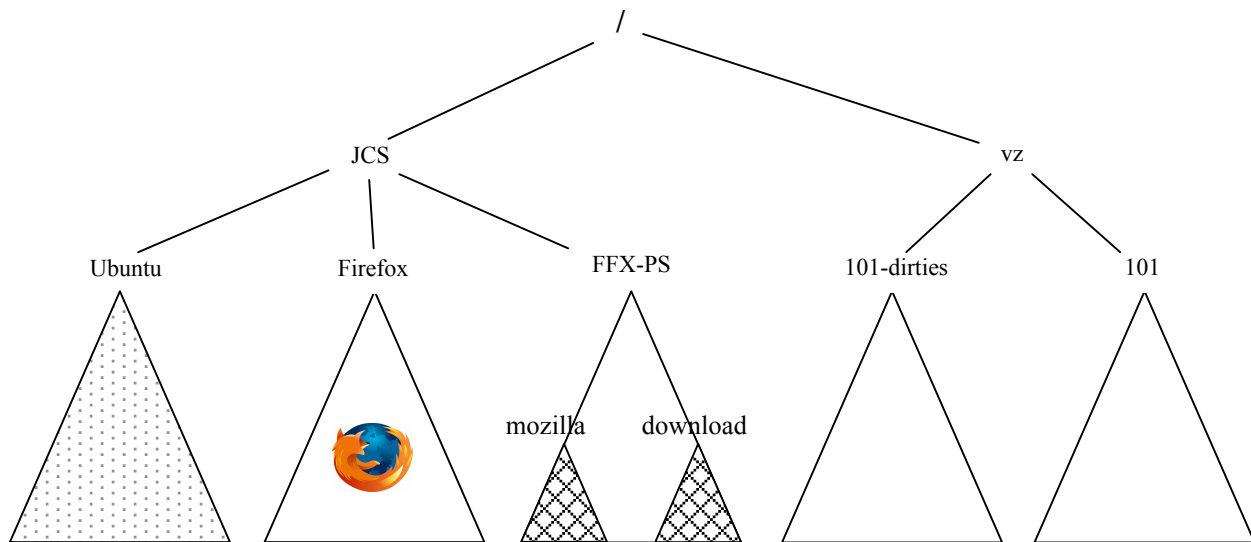


Figure 1: The directory namespaces to construct VE 101 running Firefox

Because of our unique method of constructing VEs on demand, not all virtual technologies are appropriate for JCS. Virtualization technologies can be generally cataloged as full virtualization, para-virtualization, and light-weight virtualization. With both full virtualization and para-virtualization, every virtualized environment, more commonly called a virtual machine (VM), has its own OS kernel. A VM is also configured with a virtual or physical disk to create its own file system. Host and VM file systems are managed separately by different kernels. In fact, the host may not even understand the guest file system and vice versa.

With light-weight virtualization, all VEs, also known as *containers*, share the kernel of the host system. The file system of a VE is a sub-tree of the host file system, and processes running in the VE have no access to any objects outside the sub-tree. It is this property that allows a VE's file system to be constructed by namespace unification from different parts of the host. Presently our prototype uses UnionFS [37, 38] for namespace unification and OpenVZ [36] as the light-weight virtualization technology. In general, the concept of JCS is applicable to any platform that supports light-weight virtualization and namespace unification of file systems.

3.3 Transaction Summarization

For any system monitoring and introspection solution, one critical design factor is the granularity of the events to watch. We may choose only to see a file evolve from one open-close cycle to another, equivalent to versioning on close. Alternatively we can monitor the effects of every read/write operation on the file, equivalent to versioning on writes. The general-purpose JCS framework accommodates all levels of granularity; the granularity level at which transactions are logged is to be answered by specific requirements.

On the surface, this sounds like leaving a difficult and yet critical problem to the users. To resolve the dilemma, we introduce an important contribution of this work, transaction summarization. The finest resolution of events a JCS administrator chooses to monitor and inspect produces atomic transactions (AT) over the set of variables. If we adopt the common approach of logging system calls, then each system call constitutes an AT. If we monitor the effect of every machine instruction on memory, then

every load/store instruction is an AT. In any case, an AT includes one operation on one variable.

In JCS, the collective effects of a sequence of ATs can be combined, or summarized, in a summarized transaction (ST). As an example, three atomic writes in byte intervals [0,99] at time t_1 , [100,199] at t_2 , and [200,299] at t_3 to the same file, can be summarized as one write to [0,299] from time t_1 to t_3 , combining three ATs as one ST. We note that STs are associated with an interval and generally lose information. In the above example, we lose certain time information about byte interval [1,199]. We only know it occurs between t_1 to t_3 ; the precise time t_2 is lost in summarization. In fact, we cannot deduce from the ST when byte 99 was written; the ST does not record originally there were three ATs. Lastly but not less importantly, multiple STs can be further summarized. While the above approach of summarizing consecutive identical transactions seems relatively trivial, we are also investigating more sophisticated methods that capture high-level application semantics as well as one that is lossless regarding *causality dependency* analysis.

Transaction summarization serves two important purposes. First, it allows switching event granularity on the fly. Assume that we initially model individual load/store instructions as ATs, but during peak workload find the overhead of such instrumentation too high. We can subsequently switch to page level instrumentation, producing page level STs directly. Secondly, summarization allows records in the history to fade gracefully. Assume that we model every network message exchange as ATs. This allows us to see the effect of every incoming message on the recipient VE. We will have the opportunity to see exactly which message caused a successful stack-smashing attack. Logging every message exchange, however, will quickly accumulate prodigious amounts of data. Using a period, e.g., a week, we can summarize packet exchange ATs older than a week. The summarized transactions can merely record the start time and end time of sockets and their remote endpoints (IP address and port number), leaving out details of message exchanges and drastically reducing the amount of information. If we perform this summarization daily, we can keep very long periods history of communication activities with the size of the preserved information under control. The result enables very fine-grained

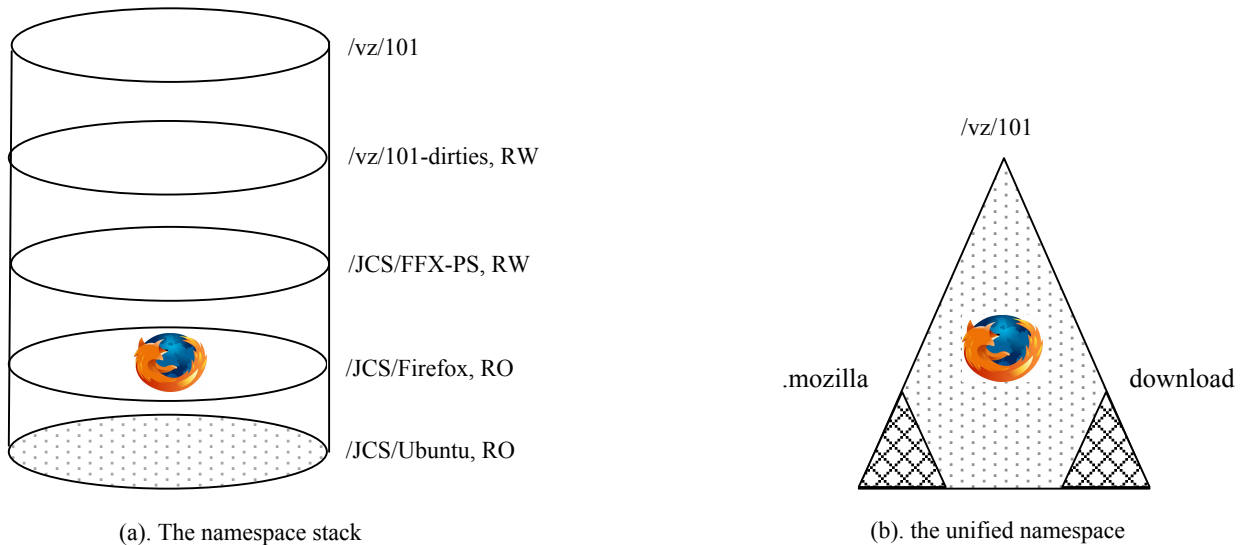


Figure 2: Unifying namespaces

intrusion analysis for recent history (within one week), and retains the ability for coarse-grained analysis much farther into the past.

It has been argued that the disk space nowadays is so big and cheap, the sizes of logs are no longer of concern. In JCS, summarizations allow preserving history for very long periods of time. A server for instance typically runs for months, if not years, continuously. The present approach of keeping log files at a manageable size is simply to cut information older than a certain threshold. With summarizations, we expect a JCS system to be able to maintain information for long periods of times while retaining the capability for precise analysis and/or system recovery. Summarizations also enable the use of JCS in space-limited systems, such as solid-storage laptops.

4. APPLICATION VIRTUAL ENVIRONMENTS USING CONTAINERS

A large body of research has been dedicated to inspecting VE activities in order to protect VE integrity. JCS, in contrast, monitors only those activities that cause interactions with other VEs and remote hosts. Due to our unique VE construction, overhead of monitoring VE file system corruption occurs only when actual corruption occurs. We present here how this unique property is achieved through a branching file namespace in each VE.

Our VE construction method leverages previous work of file namespace unifications [38]. As an example, we show in Figure 1 the directory namespaces involved in constructing a Firefox VE. As seen, a minimum Ubuntu Linux template is prebuilt at directory /JCS/Ubuntu. This directory contains bare-bone Linux utilities and libraries. The Firefox package is installed in directory /JCS/Firefox. In general, every application designated to execute in VEs will have its package installed a directory under /JCS. Both /vz/101 and /vz/101-dirties are newly created, empty directories. The Firefox application needs to preserve data after its termination. In particular, files the user downloaded from the Internet needs to be preserved. Other persistent data include browsing history, passwords and web caches, stored in a .mozilla directory. To preserve persistent data, another directory /JCS/FFX-PS (PS

stands for Persistent Storage) is created together with two subdirectories, download and .mozilla.

In Figure 2(a), we show a unified namespace /vz/101 created by stacking up four directories: /JCS/Ubuntu, /JCS/Firefox, /JCS/FFX-PS and /vz/101-dirties. Notice that /JCS/Ubuntu and /JCS/Firefox are specified as read-only namespaces, whereas /JCS/FFX-PS and /vz/101-dirties are writable. The result is shown in Figure 2(b), where one can see in the union mount directory /vz/101 all the contents of the unified namespace. Since /vz/101-dirties is empty at this point, we see in the union directory /vz/101 a minimum Ubuntu, the Firefox package, and Firefox persistent data preserved from previous sessions.

When VE 101 is booted up, /vz/101 becomes its entire file system. The Firefox browser is launched from within the VE and displayed on the desktop. For the Firefox thus executed, it is presented with a unified view of the above four directories (Figure 2(a)). It cannot see, access, distinguish or control constituent directories. Inside VE 101, all the writes to the download directory goes to /JCS/FFX-PS/download. All the writes to .mozilla directory goes to /JCS/FFX-PS/.mozilla. All other writes goes to /vz/101-dirties, named so because Firefox should not write to the rest of the system. If a Firefox vulnerability allows a Trojan horse ps program to be installed in directory /bin. It is revealed on the host as /vz/101-dirties/bin/ps on the host. In this way, the task of monitoring VE integrity is reduced to monitoring only those system activities involving /vz/101-dirties. If there is no corruption, there are no activities in /vz/101-dirties, incurring zero monitoring overhead. We will show in a later section to what (drastic) extent this approach reduces the number of system activities that have to be monitored.

In a unified namespace, constituent namespaces higher in the stack have precedence over lower ones. Following on the Trojan horse scenario, there will be two different copies of ps, the original one in /JCS/Ubuntu/bin/ps and a Trojan horse copy in /vz/101-dirties/bin/ps. Since /vz/101-dirties is at the top of the namespace stack, the VE sees the

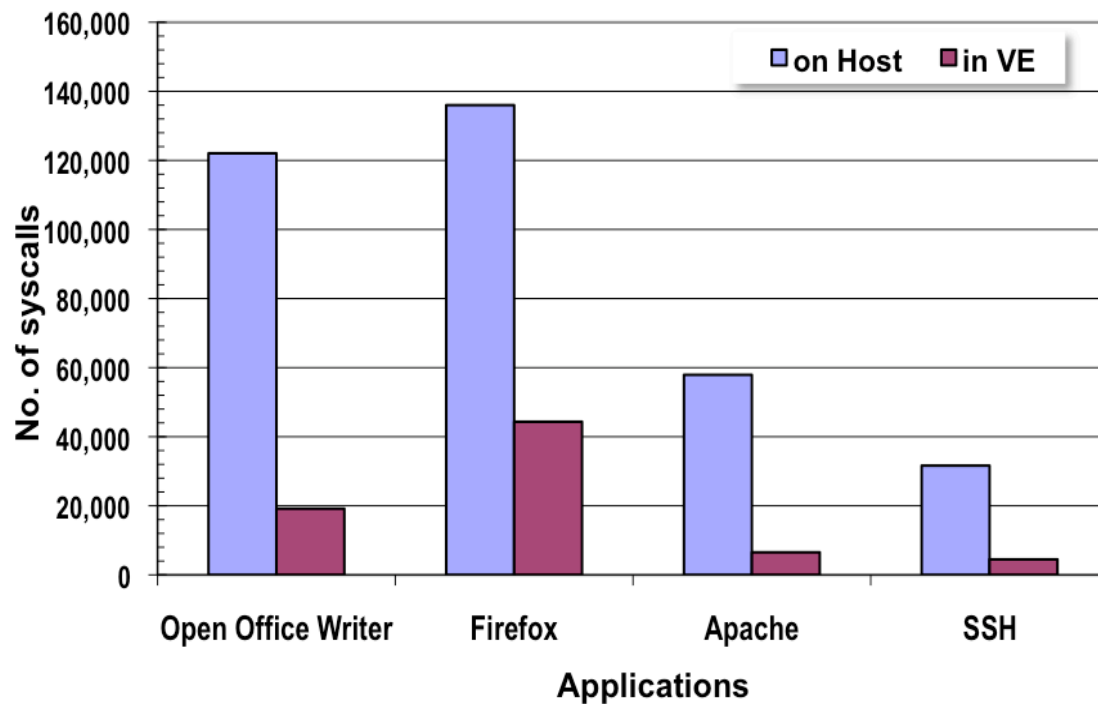


Figure 3: The number of system calls events recorded on the host versus the ones recorded in the application container for various applications. Notice that there is a significant reduction in the number of events in all cases and orders of magnitude drop in the case of Apache and SSH servers.

Trojan horse copy, while the original copy remains intact in `/JCS/Ubuntu`. When the user terminates the Firefox application, the VE is shut down and unified namespace is removed, that is, directory `/vz/101` is un-mounted and removed. Next VE will be numbered 102 and constructed as above probably with a different designated application. In this way, we always create VEs in pristine state to run newly launched applications. VEs are never reused. After application termination, directory `/vz/101-dirties` can simply be discarded or preserved for further inspection, for example, to examine the Trojan horse `ps`. We point out that certain details are left out in the above discussion, in particular, the mapping of `/JCS/FFX-PS/download` to the user's desktop.

At the first glance, the above VE construction method seems to make the tasks of maintenance and updates difficult. The opposite is true. To update Firefox, for instance, we create a VE as above, except that the `/JCS/Firefox` branch will be configured as writable. Using any package management software of choice to update (`apt-get` in Debian variants of Linux or `rpm/yum` for RedHat variants), the changes/updates will be made in the `/JCS/Firefox` branch. Subsequent Firefox VE will therefore use the updated version. The same method applies to changing configuration files and server contents. Please refer to [38] regarding policies of selecting a branch to write when more than one branch are writable.

5. Preliminary Experimental Evaluation

To evaluate the performance of our approach, we measured the amount of system call events generated by our system for popular

desktop and server applications and compared them to those of Taser (Forensix). In our prototype, we use `kprobes` as the kernel instrumentation tool to monitor and log the same set of system calls as Taser (65 in total). The main difference is that Taser was recording the system calls for the entire host operating system whereas we only focused on system calls invoked by application running inside VEs. For each application we used the same load for both systems that included typical tasks such as opening a file, editing and saving. The results presented have been verified by multiple experiments and the deviation from the numbers presented is not statistically significant. Figure 3 shows our results for a set of applications. For the Open Office application, we opened an early draft of this paper and an accompanying reference file. The early (and incomplete) draft of the paper was 207360 bytes and the reference document was 55296 bytes. We copied-and-pasted the references to the end of the draft, closed the reference file without saving, and saved/closed the draft file. The entire process lasted approximately two minutes. To measure the Firefox application, we visited our research center home page, `csis.gmu.edu`, followed every first-level links, and finally returned to the home page. The entire experiment was performed within a five-minute time interval. As a server application, we used the latest Apache version and the contents of `http://csis.gmu.edu` all running inside a container. A client machine was used to retrieve (`wget`) the home page every second. The total duration of the experiment was again 5 minutes. Finally, we tested the secure shell (server) application as follows: using the `ssh` client application, a user on a remote machine logged into the a container with access to the entire common file system including the apache configuration, and used the "more" command to go

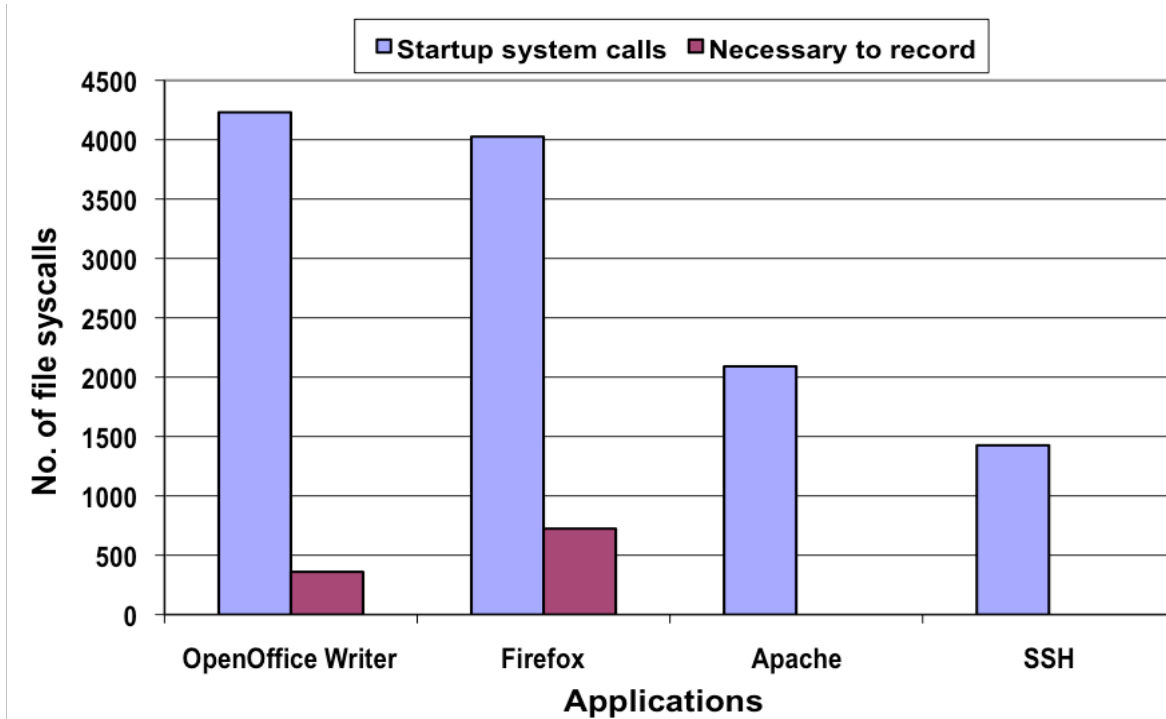


Figure 4: A comparison of file system calls generated by an application startup versus the ones that we are required to record because they are either not part of the startup or they generate new files including temporary files. Both apache and ssh do not require any new files upon startup and thus they do not require any startup logging as expected.

through the entire apache configuration file (a few kilobytes). Both apache and ssh servers resided in containers with identical setup and sshd service was disabled in the Apache container and vice versa. Therefore, in all experiments, a container runs only those processes and threads belonging to the designated application (*i.e. there are no (other) background daemons running concurrently*).

We also noticed that there is a significant drop in the amount of events that we need to record when we focus on a specific application versus the entire operating system. This is an expected result because by placing the application inside a container we focus on the system calls that are generated by that specific application including all processes and threads. On the other hand, the entire operating system is more “noisy” ending up increasing the amount of events generated by many times and in some cases by orders of magnitude. That is especially the case for server applications such as Apache and SSH.

In the next experiment, we measured the numbers of events that need to be recorded in JCS when launching applications. When an application is launched, it first loads its own binary itself and subsequently required libraries. The firefox in particular also reads language packs and pre-built javascripts. Those files are part of the read-only branches (as described in Section 4), created when installing the package branch. Since all JCS containers start in pristine state, system calls on those files can be safely ignored before the applications communicates with other VEs or the outside world. However, it is sometimes difficult to define or capture the exact point when an application transits from the startup process to normal operation phase. For the Apache and secure shell applications, the startup activities are easy to observe --- after

startup, there will be no system activities without client requests. We simply started the application, waited for a couple of seconds, and then created one client request. All activities before the handling of the request are considered startup activities. Examinations on the log files clearly confirm the existence of a quite period before handling the request. For the Firefox application, we set the default home page to none. We stopped logging when we saw the Firefox application showing up on the desktop with a blank page. For the Open Office application, we launch it through command line without providing a file name. We stopped logging when the Office window showed up with blank content. For Open Office, the user preference settings are stored in a shared directory to keep previous settings. For Firefox, user preferences, cookies, browsing history, web caches and stored passwords are stored in a shared directory. Accessing data in any shared directory must be recorded for they may include contamination or corruptions left by the previous sections. Apache configuration and web contents are however considered static and are part of the container (in a read-only branch). The same apply the ssh application; hence the zero activities necessary to record in Figure 4. We consider changes in server configuration, contents and the application/library binaries as a maintenance and update operation, a subject discussed previously.

By using known good but potential vulnerable binaries from a UnionFS read-only slice, we can reduce the number of recorded events even further (see Figure 4). Notice that although the drop in events might not seem significant compared to the events recorded in Figure 3, they have a cumulative effect over many instances of applications. For instance, if we startup Firefox 10 times over the period of monitoring, our system will record

around 7,500 system calls whereas Taser will record 42,000 events an approximate 1 over 5 ratio.

6. CONCLUSION

We presented a framework for tracking application interactions using a lightweight virtualization tool combined with a stackable file system. Our goal was to reduce the amount of system calls recorded for each application while maintaining full tracking information of the application interactions. To that end, we introduced an architecture that can scale with the number of applications and record data over a large period of time. Our approach maintains application integrity and offers the ability to utilize the recorded information for a wide range of other uses. This includes system recovery and information tracking among others. Although this is the first step, we believe our approach to be the first that can provide a practical and efficient approach to application tracking and a potential mechanism to record all system events as database transactions without preventive storage and monitoring overhead.

7. REFERENCES

- [1] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "VMM-based hidden process detection and identification using Lycosid," In ACM Conference on Virtual Execution Environments (VEE), Seattle, WA, Mar. 2008.
- [2] X. Jiang, X. Wang, and D. Xu. "Stealthy malware detection through VMM-based 'out-of-the-box' semantic view," In 14th ACM Conference on Computer and Communications Security (CCS), Alexandria, VA, Nov. 2007.
- [3] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. van Doorn. "Building a MAC-based security architecture for the Xen open-source hypervisor," In Annual Computer Security Applications Conference (ACSAC), Tucson, AZ, Dec. 2005.
- [4] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2003.
- [5] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. 2002. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. SIGOPS Oper. Syst. Rev. 36, SI (Dec. 2002), 211-224.
- [6] King, S. T. and Chen, P. M. 2005. Backtracking intrusions. ACM Trans. Comput. Syst. 23, 1 (Feb. 2005), 51-76.
- [7] Payne, B.D.; de Carbone, M.D.P.; Wenke Lee, "Secure and Flexible Monitoring of Virtual Machines," Annual Computer Security Applications Conference (ACSAC), 2007. pp.385-397, Dec. 2007.
- [8] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Antfarm: Tracking processes in a virtual machine environment," In USENIX Annual Technical Conference, Boston, MA, June 2006.
- [9] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing, NY, Oct. 2003.
- [10] Samuel T. King, George W. Dunlap, and Peter M. Chen. "Debugging operating systems with time-traveling virtual machines," In Proceedings of Unenix '05, Anaheim, CA, April 2005.
- [11] Laadan, O., Baratto, R. A., Phung, D. B., Potter, S., and Nieh, J. "DejaView: a personal virtual computer recorder" In Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, Stevenson, Washington, USA, October 2007.
- [12] X. Jiang and X. Wang. "Out-of-the-box monitoring of VM-based high-interaction honeypots," In 10th International Symposium on Recent Advances in Intrusion Detection (RAID), Surfers Paradise, Australia, Sept. 2007.
- [13] Jiang, X. and Xu, D. 2004. "Collapsar: a VM-based architecture for network attack detention center," In Proceedings of the 13th Conference on USENIX Security Symposium, San Diego, CA, August 2004.
- [14] Vrabie, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A. C., Voelker, G. M., and Savage, S. "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," SIGOPS Oper. Syst. Rev. 39, 5 (Oct. 2005), 148-162.
- [15] Asrigo, K., Litty, L., and Lie, D. "Using VMM-based sensors to monitor honeypots," In Proceedings of the 2nd international Conference on Virtual Execution Environments, Ottawa, Ontario, Canada, June 2006.
- [16] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In Proceeding of the 16th USENIX Security Symposium, August 2007.
- [17] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. "A sense of self for UNIX processes," In IEEE Symposium on Security and Privacy, Oakland, CA, May 1996.
- [18] Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji. Intrusion detection using sequences of system calls. Journal of Computer Security. Vol. 6, No. 3, 1998, Pages 151—180.
- [19] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. "Exploiting execution context for the detection of anomalous system calls," In 10th International Symposium on Recent Advances in Intrusion Detection (RAID), Surfers Paradise, Australia, Sept. 2007.
- [20] J. T. Giffin, S. Jha, and B. P. Miller. "Efficient context-sensitive intrusion detection," In Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2004.
- [21] Sitaraman, S.; Venkatesan, S., "Forensic analysis of file system intrusions using improved backtracking," Information Assurance, Proceedings. Third IEEE International Workshop on, pp. 154-163, March 2005.
- [22] Goel, A.; Feng, W.-C.; Maier, D.; Walpole, J., "Forensix: a robust, high-performance reconstruction system," 25th IEEE International Conference on Distributed Computing Systems Workshops, pp. 155-162, June 2005.
- [23] Goel, A., Feng, W., Feng, W., and Maier, D. Automatic high-performance reconstruction and recovery. Computer Networks. 51, 5 (Apr. 2007), 1361-1377.

- [24] Goel, A., Po, K., Farhadi, K., Li, Z., and de Lara, E. "The taser intrusion recovery system," In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05). Brighton, United Kingdom, pp. 163-176., October, 2005.
- [25] Qin, F., J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies – a safe method to survive software failures," in Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP), pp. 235-248, 2005.
- [26] R. J. Creasy. The origin of the VM/370 time-sharing system. IBM J. Research and Development, 25(5):483-490, September 1981.
- [27] VMware, <http://www.vmware.com>.
- [28] Paul Barham , Boris Dragovic , Keir Fraser , Steven Hand , Tim Harris , Alex Ho , Rolf Neugebauer , Ian Pratt , Andrew Warfield, "Xen and the Art of Virtualization," Proceedings of the nineteenth ACM symposium on Operating systems principles, October 19-22, 2003, Bolton Landing, NY, USA.
- [29] Jeff Dike, "A User-Mode Port of the Linux Kernel," Proceedings of the 2000 Linux Showcase and Conference, October 2000.
- [30] KVM (Kernel-based Virtual Machine). http://kvm.qumranet.com/kvmwiki/Front_Page.
- [31] Jon Watson. VirtualBox: bits and bytes masquerading as machines. Linux Journal, Volume 2008 , Issue 166, February 2008.
- [32] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In Proc. of 2nd Intl. SANE Conference, May 2000.
- [33] Soltesz, S., Pötl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," In Proceedings of the ACM Sigops/Eurosys European Conference on Computer Systems (EuroSys '07), Lisbon, Portugal, March 2007.
- [34] Osman, S., Subhraveti, D., Su, G., and Nieh, J. 2002. The design and implementation of Zap: a system for migrating computing environments. In Proceedings of the 5th Symposium on Operating Systems Design and Implementation. (Boston, Massachusetts, December 09 - 11, 2002). OSDI '02.
- [35] Price, D. and Tucker, A. 2004. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In Proceedings of the 18th USENIX Conference on System Administration (Atlanta, GA, November 14 - 19, 2004).
- [36] <http://openvz.org>: OpenVZ server virtualization open-source project.
- [37] Wright, C. P., Dave, J., Gupta, P., Krishnan, H., Quigley, D. P., Zadok, E., and Zubair, M. N. 2006. Versatility and Unix semantics in namespace unification. Trans. Storage 2, 1 (Feb. 2006), 74-105.
- [38] Unionfs: <http://www.am-utils.org/project-unionfs.html>